VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
On

OPERATING SYSTEMS (23CS4PCOPS)

Submitted by

SHASHANK U (1BM23CS314)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
February-May 2025

B. M. S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

This is to certify that the Lab work entitled **“OPERATING SYSTEMS”** carried out by Shashank U**(1BM23CS314)**, who is bonafide student of **B. M. S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2024-25. The Lab report has been approved as it satisfies the academic requirements in respect of Operating Systems Lab - **(23CS4PCOPS)** work prescribed for the said degree.

Prof. Amruta	**Dr. Kavitha Sooda**
Assistant Professor	Professor and Head
Department of CSE	Department of CSE
BMSCE, Bengaluru	BMSCE, Bengaluru

Index Sheet

Course Outcomes (COs):

CO1	Apply the different concepts and functionalities of Operating System.
CO2	Analyse various Operating system strategies and techniques.
CO3	Demonstrate the different functionalities of Operating System.
CO4	Conduct practical experiments to implement the functionalities of Operating system.

Lab program 1:

Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time.

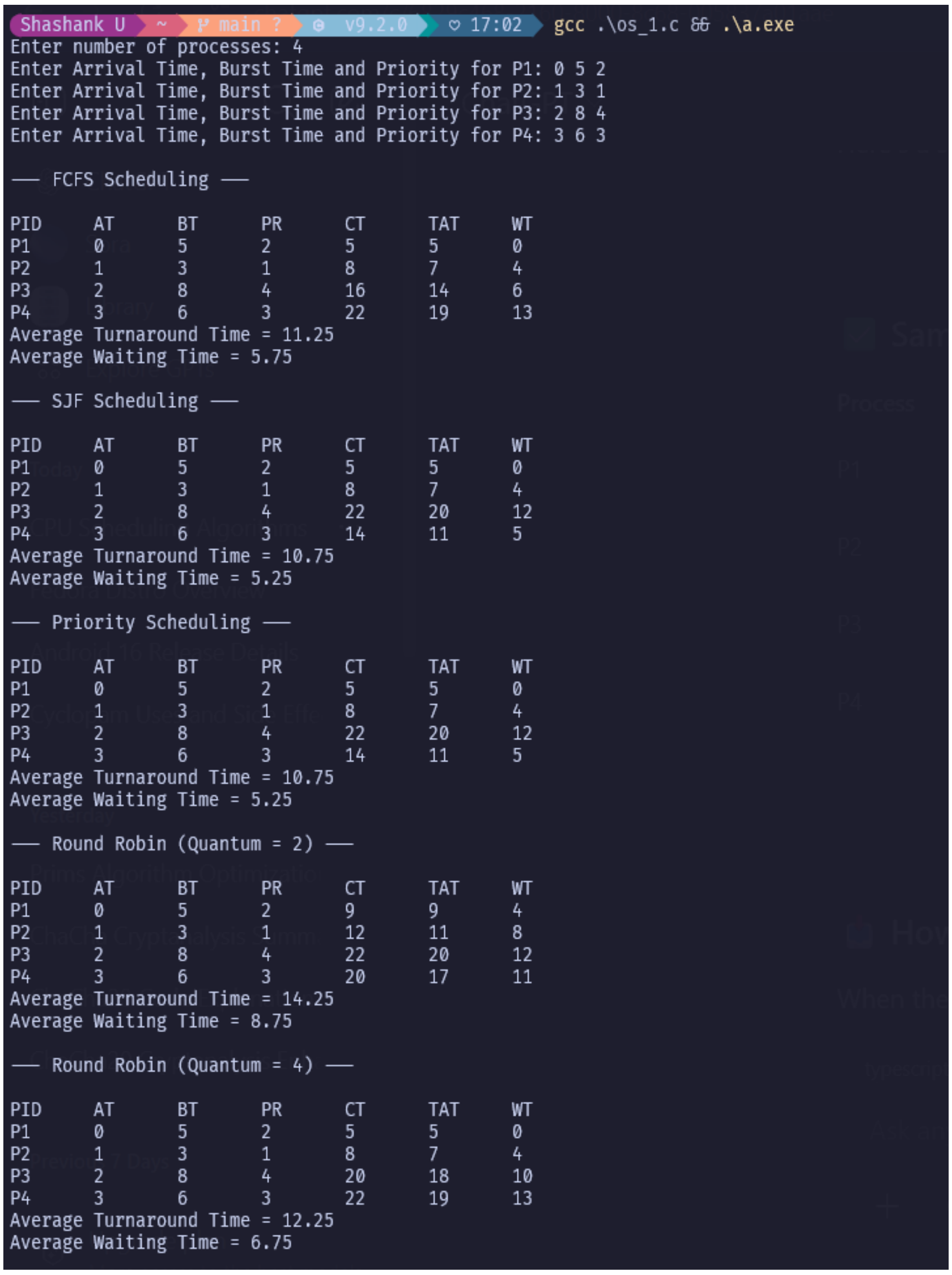
a) FCFS b) SJF c) Priority d) Round Robin (Experiment with different quantum sizes for RR algorithm)

```
#include <stdio.h>

#define MAX 100

typedef struct {

    int pid, arrival, burst, burst_copy, priority, completion, turnaround, waiting, finished;

} Process;


void inputProcesses(Process p[], int *n) {

    printf("Enter number of processes: ");

    scanf("%d", n);

    for (int i = 0; i < *n; i++) {

        p[i].pid = i + 1;

        printf("Enter Arrival Time, Burst Time and Priority for P%d: ", i + 1);

        scanf("%d %d %d", &p[i].arrival, &p[i].burst, &p[i].priority);

        p[i].burst_copy = p[i].burst;

        p[i].finished = 0;

    }

}


void displayResults(Process p[], int n) {

    float total_TAT = 0, total_WT = 0;

    printf("\nPID\tAT\tBT\tPR\tCT\tTAT\tWT\n");

    for (int i = 0; i < n; i++) {

        p[i].turnaround = p[i].completion - p[i].arrival;

        p[i].waiting = p[i].turnaround - p[i].burst_copy;

        total_TAT += p[i].turnaround;

        total_WT += p[i].waiting;
        printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].arrival, p[i].burst_copy, p[i].priority,
               p[i].completion, p[i].turnaround, p[i].waiting);
    }
    printf("Average Turnaround Time = %.2f\n", total_TAT / n);
    printf("Average Waiting Time = %.2f\n", total_WT / n);
}

void FCFS(Process p[], int n) {
    Process temp[MAX];
    for (int i = 0; i < n; i++) temp[i] = p[i];

    for (int i = 0; i < n - 1; i++)  // Sort by arrival time
        for (int j = 0; j < n - i - 1; j++)
            if (temp[j].arrival > temp[j + 1].arrival) {
                Process t = temp[j];
                temp[j] = temp[j + 1];
                temp[j + 1] = t;
            }

    int time = 0;
    for (int i = 0; i < n; i++) {
        if (time < temp[i].arrival)
            time = temp[i].arrival;
        time += temp[i].burst;
        temp[i].completion = time;
    }

    printf("\n--- FCFS Scheduling ---\n");
    displayResults(temp, n);
}

void SJF(Process p[], int n) {
    Process temp[MAX];
    for (int i = 0; i < n; i++) temp[i] = p[i];

    int completed = 0, time = 0;
    while (completed < n) {
        int idx = -1, minBT = 1e9;
        for (int i = 0; i < n; i++) {
            if (!temp[i].finished && temp[i].arrival <= time && temp[i].burst < minBT) {
                minBT = temp[i].burst;
                idx = i;
            }
        }
        if (idx != -1) {
            time += temp[idx].burst;
            temp[idx].completion = time;
            temp[idx].finished = 1;
            completed++;
        } else {
            time++;
        }
    }

    printf("\n--- SJF Scheduling ---\n");
    displayResults(temp, n);
}

void PriorityScheduling(Process p[], int n) {
    Process temp[MAX];
    for (int i = 0; i < n; i++) temp[i] = p[i];

    int completed = 0, time = 0;
    while (completed < n) {
        int idx = -1, highest = 1e9;
        for (int i = 0; i < n; i++) {
            if (!temp[i].finished && temp[i].arrival <= time && temp[i].priority < highest) {
                highest = temp[i].priority;
                idx = i;
            }
        }
        if (idx != -1) {
            time += temp[idx].burst;
            temp[idx].completion = time;
            temp[idx].finished = 1;
            completed++;
        } else {
            time++;
        }
    }

    printf("\n--- Priority Scheduling ---\n");
    displayResults(temp, n);
}

void RoundRobin(Process p[], int n, int quantum) {
    Process temp[MAX];
    for (int i = 0; i < n; i++) {
        temp[i] = p[i];
        temp[i].burst = p[i].burst_copy;
        temp[i].finished = 0;
    }

    int time = 0, completed = 0;
    int queue[MAX], front = 0, rear = 0, inQueue[MAX] = {0};
```

```
queue[rear++] = 0;
inQueue[0] = 1;

while (completed < n) {
    int found = 0;
    int size = rear - front;
    for (int i = 0; i < size; i++) {
        int idx = queue[front++];
        found = 1;

        if (temp[idx].burst <= quantum) {
            time = (time < temp[idx].arrival) ? temp[idx].arrival : time;
            time += temp[idx].burst;
            temp[idx].completion = time;
            temp[idx].burst = 0;
            temp[idx].finished = 1;
            completed++;
        } else {
            time = (time < temp[idx].arrival) ? temp[idx].arrival : time;
            time += quantum;
            temp[idx].burst -= quantum;
            queue[rear++] = idx;
        }

        for (int j = 0; j < n; j++) {
            if (!inQueue[j] && temp[j].arrival <= time && temp[j].burst > 0) {
                queue[rear++] = j;
                inQueue[j] = 1;
            }
        }
    }
```

```
        if (!found) time++;
    }

    printf("\n--- Round Robin (Quantum = %d) ---\n", quantum);
    displayResults(temp, n);
}

int main() {
    Process p[MAX];
    int n;
    inputProcesses(p, &n);

    FCFS(p, n);
    SJF(p, n);
    PriorityScheduling(p, n);

    int q1 = 2, q2 = 4;
    RoundRobin(p, n, q1);
    RoundRobin(p, n, q2);

    return 0;
}
```

OUTPUT:



```
Shashank U  ~  main ?  v9.2.0  17:02  gcc .\os_1.c && .\a.exe
Enter number of processes: 4
Enter Arrival Time, Burst Time and Priority for P1: 0 5 2
Enter Arrival Time, Burst Time and Priority for P2: 1 3 1
Enter Arrival Time, Burst Time and Priority for P3: 2 8 4
Enter Arrival Time, Burst Time and Priority for P4: 3 6 3

—— FCFS Scheduling ——

PID     AT      BT      PR      CT      TAT     WT
P1      0       5       2       5       5       0
P2      1       3       1       8       7       4
P3      2       8       4       16      14      6
P4      3       6       3       22      19      13
Average Turnaround Time = 11.25
Average Waiting Time = 5.75

—— SJF Scheduling ——

PID     AT      BT      PR      CT      TAT     WT
P1      0       5       2       5       5       0
P2      1       3       1       8       7       4
P3      2       8       4       22      20      12
P4      3       6       3       14      11      5
Average Turnaround Time = 10.75
Average Waiting Time = 5.25

—— Priority Scheduling ——

PID     AT      BT      PR      CT      TAT     WT
P1      0       5       2       5       5       0
P2      1       3       1       8       7       4
P3      2       8       4       22      20      12
P4      3       6       3       14      11      5
Average Turnaround Time = 10.75
Average Waiting Time = 5.25

—— Round Robin (Quantum = 2) ——

PID     AT      BT      PR      CT      TAT     WT
P1      0       5       2       9       9       4
P2      1       3       1       12      11      8
P3      2       8       4       22      20      12
P4      3       6       3       20      17      11
Average Turnaround Time = 14.25
Average Waiting Time = 8.75

—— Round Robin (Quantum = 4) ——

PID     AT      BT      PR      CT      TAT     WT
P1      0       5       2       5       5       0
P2      1       3       1       8       7       4
P3      2       8       4       20      18      10
P4      3       6       3       22      19      13
Average Turnaround Time = 12.25
Average Waiting Time = 6.75
```

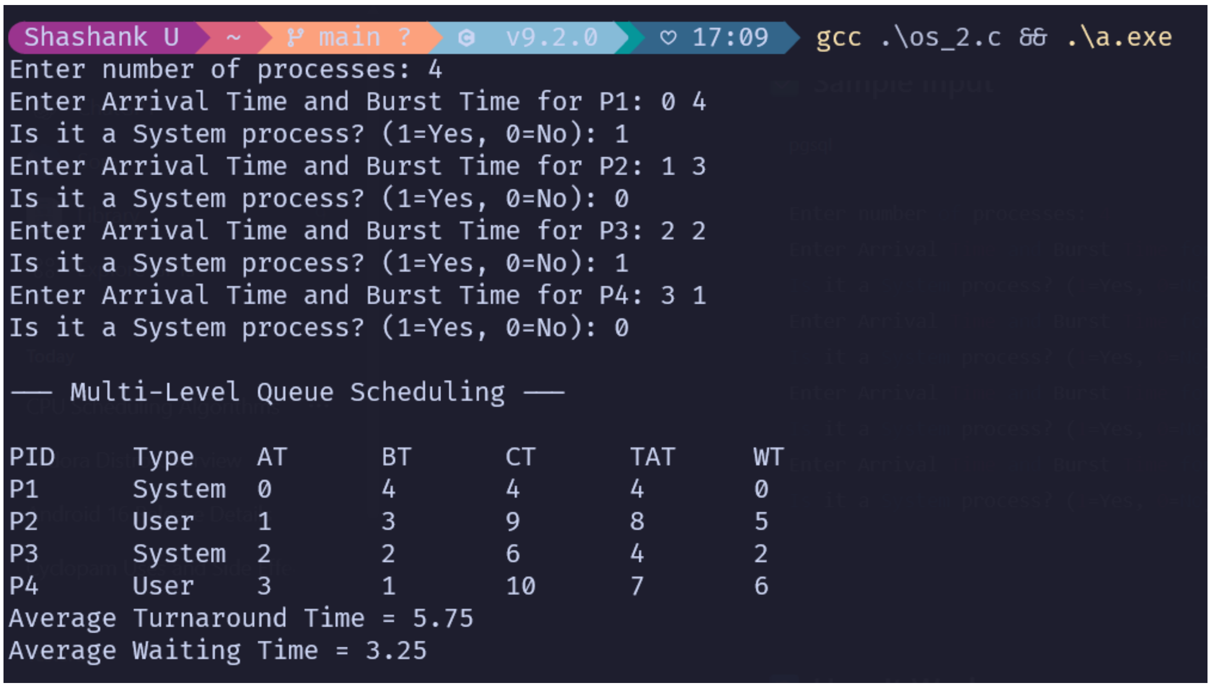
Lab Program 2:

Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue.

```
#include <stdio.h>

#define MAX 100

typedef struct {
    int pid;
    int arrival;
    int burst;
    int completion;
    int turnaround;
    int waiting;
    int isSystem;  // 1 = System process, 0 = User process
    int scheduled;
} Process;

void inputProcesses(Process p[], int *n) {
    printf("Enter number of processes: ");
    scanf("%d", n);
    for (int i = 0; i < *n; i++) {
        p[i].pid = i + 1;
        printf("Enter Arrival Time and Burst Time for P%d: ", p[i].pid);
        scanf("%d %d", &p[i].arrival, &p[i].burst);
        printf("Is it a System process? (1=Yes, 0=No): ");
        scanf("%d", &p[i].isSystem);
        p[i].scheduled = 0;
    }
}
```

```
void displayResults(Process p[], int n) {
    float total_TAT = 0, total_WT = 0;
    printf("\nPID\tType\tAT\tBT\tCT\tTAT\tWT\n");
    for (int i = 0; i < n; i++) {
        p[i].turnaround = p[i].completion - p[i].arrival;
        p[i].waiting = p[i].turnaround - p[i].burst;
        total_TAT += p[i].turnaround;
        total_WT += p[i].waiting;
        printf("P%d\t%s\t%d\t%d\t%d\t%d\t%d\n", p[i].pid,
               p[i].isSystem ? "System" : "User",
               p[i].arrival, p[i].burst,
               p[i].completion, p[i].turnaround, p[i].waiting);
    }
    printf("Average Turnaround Time = %.2f\n", total_TAT / n);
    printf("Average Waiting Time = %.2f\n", total_WT / n);
}

void multiLevelQueueScheduling(Process p[], int n) {
    int time = 0, completed = 0;

    while (completed < n) {
        int idx = -1;

        // First: look for system process that has arrived
        for (int i = 0; i < n; i++) {
            if (!p[i].scheduled && p[i].isSystem && p[i].arrival <= time) {
                idx = i;
                break;  // FCFS: pick first eligible
            }
        }
        // If no system process, check user processes
        if (idx == -1) {
            for (int i = 0; i < n; i++) {
                if (!p[i].scheduled && !p[i].isSystem && p[i].arrival <= time) {
                    idx = i;
                    break;
                }
            }
        }

        // If a process is ready to run
        if (idx != -1) {
            if (time < p[idx].arrival)
                time = p[idx].arrival;
            time += p[idx].burst;
            p[idx].completion = time;
            p[idx].scheduled = 1;
            completed++;
        } else {
            time++;  // No process is ready, so advance time
        }
    }

    printf("\n--- Multi-Level Queue Scheduling ---\n");
    displayResults(p, n);
}

int main() {
    Process p[MAX];
    int n;
    inputProcesses(p, &n);
    multiLevelQueueScheduling(p, n);
    return 0;
}
```

OUTPUT:



```
Shashank U  ~  main ?  v9.2.0  17:09  gcc .\os_2.c && .\a.exe
Enter number of processes: 4
Enter Arrival Time and Burst Time for P1: 0 4
Is it a System process? (1=Yes, 0=No): 1
Enter Arrival Time and Burst Time for P2: 1 3
Is it a System process? (1=Yes, 0=No): 0
Enter Arrival Time and Burst Time for P3: 2 2
Is it a System process? (1=Yes, 0=No): 1
Enter Arrival Time and Burst Time for P4: 3 1
Is it a System process? (1=Yes, 0=No): 0

—— Multi-Level Queue Scheduling ——

PID     Type    AT      BT      CT      TAT     WT
P1      System  0       4       4       4       0
P2      User    1       3       9       8       5
P3      System  2       2       6       4       2
P4      User    3       1       10      7       6
Average Turnaround Time = 5.75
Average Waiting Time = 3.25
```

Lab Program 3:

Write a C program to simulate Real-Time CPU Scheduling algorithms:

a) Rate- Monotonic b) Earliest-deadline First c) Proportional scheduling

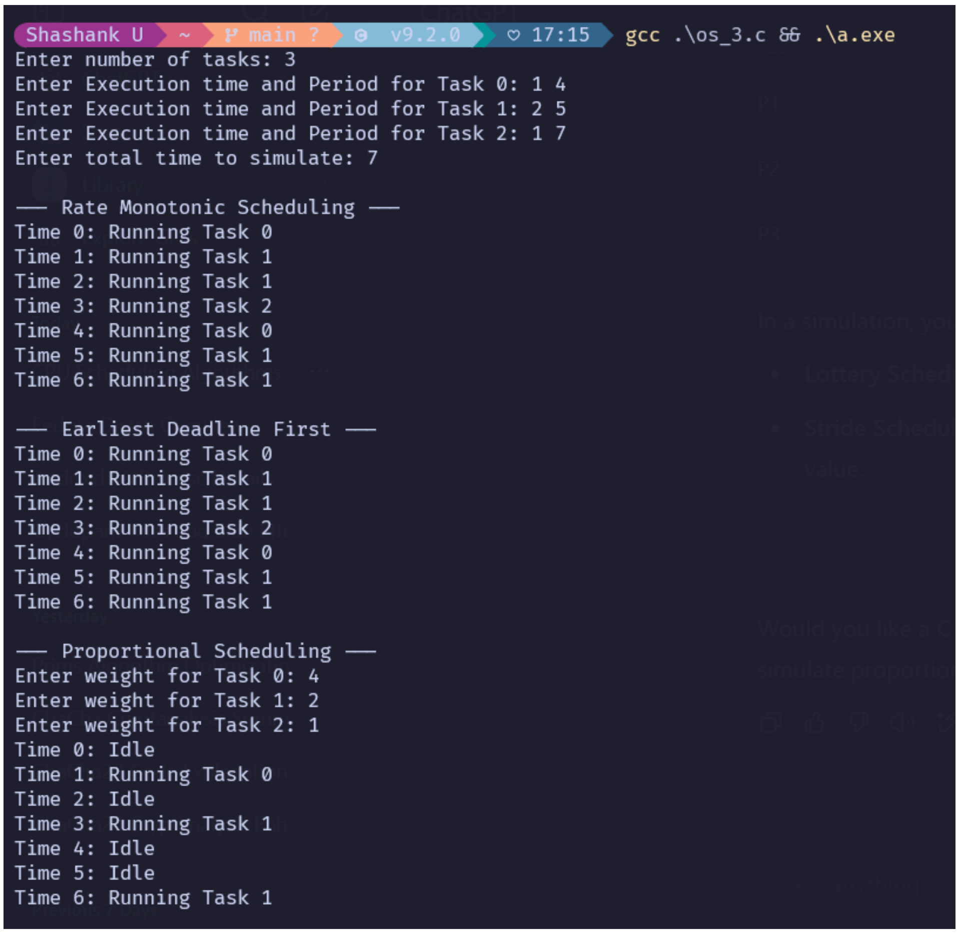
```
#include<stdio.h>
#include<stdlib.h>

typedef struct {
        int id;
        int execution_time;
        int period; // also deadline
        int remaining_time;
        int deadline;
        int weight; // for proportional scheduling
} Task;
```

```
}

void input_tasks(Task *tasks, int n){
        for(int i=0;i<n;i++){
                printf("Enter Execution time and Period for Task %d: ", i);
                scanf("%d%d", &tasks[i].execution_time, &tasks[i].period);
                tasks[i].id = i;
                tasks[i].remaining_time = tasks[i].execution_time;
                tasks[i].deadline = tasks[i].period;
        }
}

void rate_monotonic(Task *tasks, int n, int time_limit){
        printf("\n--- Rate Monotonic Scheduling ---\n");
        for(int t=0; t<time_limit; t++){
                int min_period = 9999, sel = -1;
                for(int i=0;i<n;i++){
                        if(t % tasks[i].period == 0){
                                tasks[i].remaining_time = tasks[i].execution_time;
                        }
                        if(tasks[i].remaining_time > 0 && tasks[i].period < min_period){
                                min_period = tasks[i].period;
                                sel = i;
                        }
                }
                if(sel != -1){
                        printf("Time %d: Running Task %d\n", t, sel);
                        tasks[sel].remaining_time--;
                }else{
                        printf("Time %d: Idle\n", t);
                }
        }
}
```

```
void earliest_deadline_first(Task *tasks, int n, int time_limit){
        printf("\n--- Earliest Deadline First ---\n");
        for(int i=0;i<n;i++){
                tasks[i].deadline = tasks[i].period;
        }
        for(int t=0;t<time_limit;t++){
                int min_deadline = 9999, sel = -1;
                for(int i=0;i<n;i++){
                        if(t % tasks[i].period == 0){
                                tasks[i].remaining_time = tasks[i].execution_time;
                                tasks[i].deadline = t + tasks[i].period;
                        }
                        if(tasks[i].remaining_time > 0 && tasks[i].deadline < min_deadline){
                                min_deadline = tasks[i].deadline;
                                sel = i;
                        }
                }
                if(sel != -1){
                        printf("Time %d: Running Task %d\n", t, sel);
                        tasks[sel].remaining_time--;
                }else{
                        printf("Time %d: Idle\n", t);
                }
        }
}

void proportional_scheduling(Task *tasks, int n, int time_limit){
        printf("\n--- Proportional Scheduling ---\n");
        int total_weight = 0;
        for(int i=0;i<n;i++){
                printf("Enter weight for Task %d: ", i);
                scanf("%d", &tasks[i].weight);
```

```
            total_weight += tasks[i].weight;
            tasks[i].remaining_time = 0;
        }
        int allocated[100] = {0};
        for(int t=0;t<time_limit;t++){
            int sel = -1;
            int max_ratio = -1;
            for(int i=0;i<n;i++){
                if(tasks[i].execution_time > 0){
                    int expected = (tasks[i].weight * (t+1)) / total_weight;
                    if(allocated[i] < expected && expected - allocated[i] > max_ratio){
                        max_ratio = expected - allocated[i];
                        sel = i;
                    }
                }
            }
            if(sel != -1){
                printf("Time %d: Running Task %d\n", t, sel);
                allocated[sel]++;
                tasks[sel].execution_time--;
            }else{
                printf("Time %d: Idle\n", t);
            }
        }
}

int main(){
        int n, time_limit;
        printf("Enter number of tasks: ");
        scanf("%d", &n);
        Task tasks[10], tasks_copy[10];
        input_tasks(tasks, n);

        printf("Enter total time to simulate: ");
        scanf("%d", &time_limit);

        // RMS
        for(int i=0;i<n;i++) tasks_copy[i] = tasks[i];
        rate_monotonic(tasks_copy, n, time_limit);

        // EDF
        for(int i=0;i<n;i++) tasks_copy[i] = tasks[i];
        earliest_deadline_first(tasks_copy, n, time_limit);

        // Proportional
        for(int i=0;i<n;i++) tasks_copy[i] = tasks[i];
        proportional_scheduling(tasks_copy, n, time_limit);

        return 0;
}
```

OUTPUT:



```
Shashank U  ~  main ?  v9.2.0  17:15  gcc .\os_3.c && .\a.exe
Enter number of tasks: 3
Enter Execution time and Period for Task 0: 1 4
Enter Execution time and Period for Task 1: 2 5
Enter Execution time and Period for Task 2: 1 7
Enter total time to simulate: 7

—— Rate Monotonic Scheduling ——
Time 0: Running Task 0
Time 1: Running Task 1
Time 2: Running Task 1
Time 3: Running Task 2
Time 4: Running Task 0
Time 5: Running Task 1
Time 6: Running Task 1

—— Earliest Deadline First ——
Time 0: Running Task 0
Time 1: Running Task 1
Time 2: Running Task 1
Time 3: Running Task 2
Time 4: Running Task 0
Time 5: Running Task 1
Time 6: Running Task 1

—— Proportional Scheduling ——
Enter weight for Task 0: 4
Enter weight for Task 1: 2
Enter weight for Task 2: 1
Time 0: Idle
Time 1: Running Task 0
Time 2: Idle
Time 3: Running Task 1
Time 4: Idle
Time 5: Idle
Time 6: Running Task 1
```

Lab Program 4

Write a C program to simulate: a) Producer-Consumer problem using semaphores.

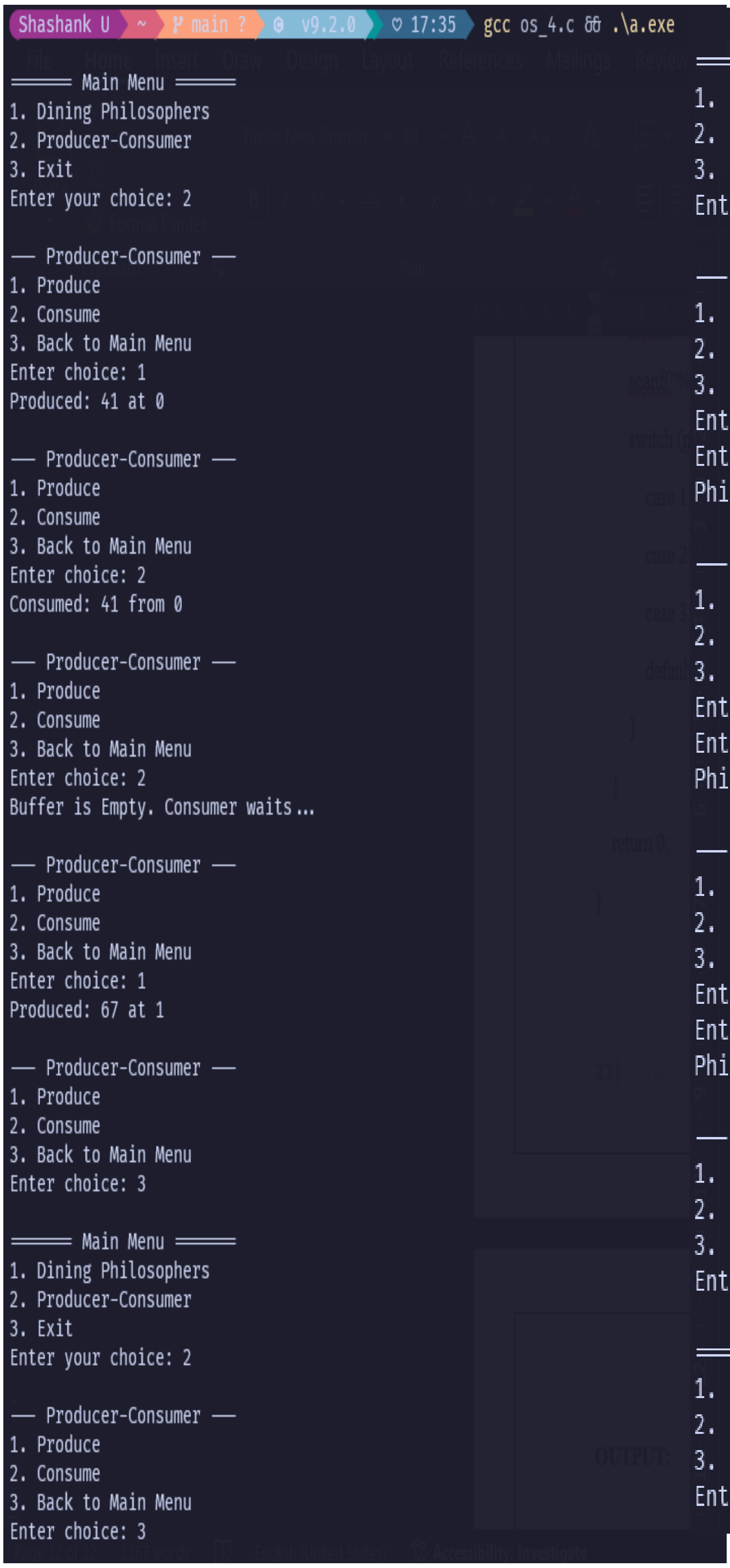
b) Dining-Philosopher’s problem

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>


// ---------- Dining Philosophers ----------

int dp_state[5]; // 0: Thinking, 1: Hungry, 2: Eating

int forks[5] = {1, 1, 1, 1, 1}; // 1 means fork is available


void dp_take_forks(int p) {

    if (forks[p] && forks[(p + 1) % 5]) {

        forks[p] = forks[(p + 1) % 5] = 0;

        dp_state[p] = 2;

        printf("Philosopher %d is Eating\n", p);

    } else {

        dp_state[p] = 1;

        printf("Philosopher %d is Hungry but forks not available\n", p);

    }

}


void dp_put_forks(int p) {

    if (dp_state[p] == 2) {

        forks[p] = forks[(p + 1) % 5] = 1;

        dp_state[p] = 0;

        printf("Philosopher %d puts down forks and starts Thinking\n", p);

    } else {

        printf("Philosopher %d was not Eating\n", p);

    }

}
```

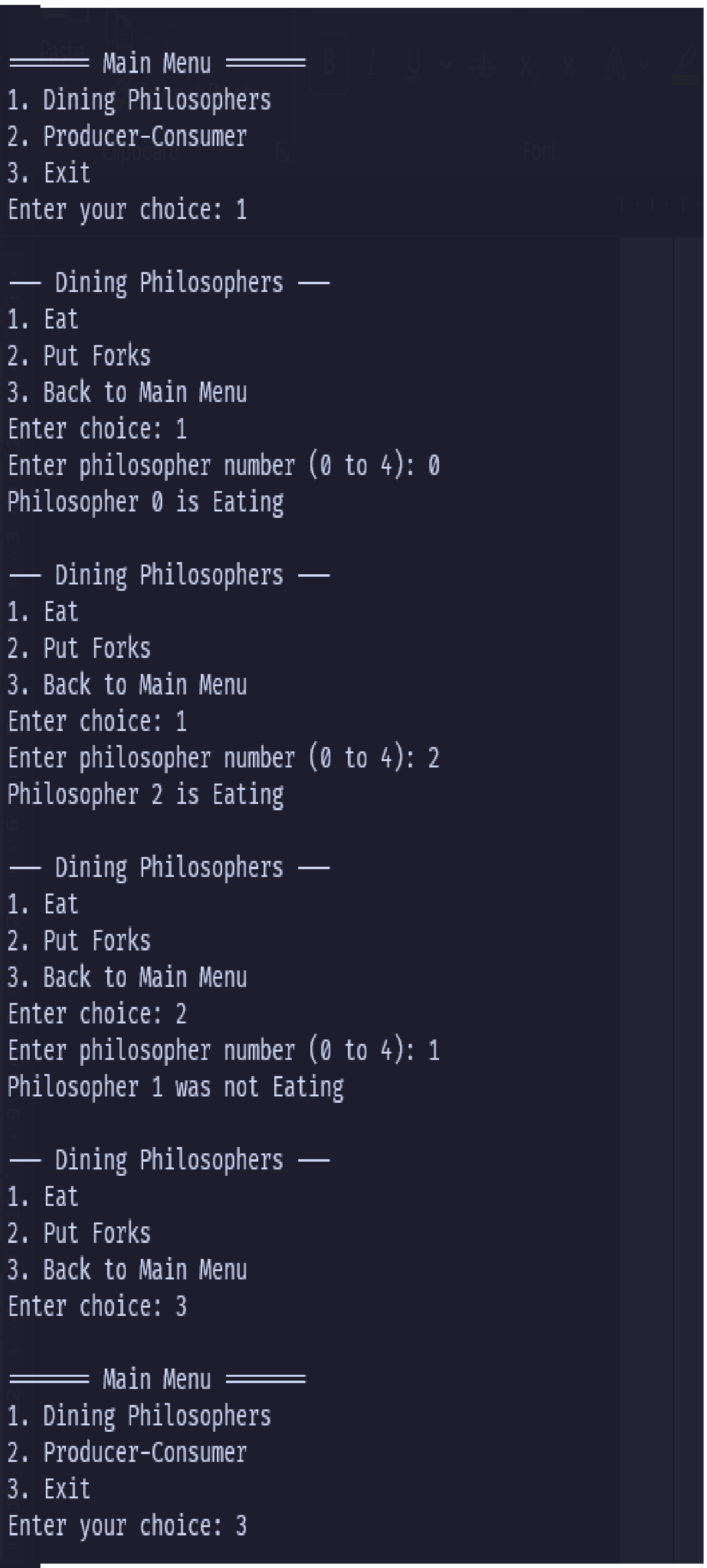
```
void run_dining_philosophers() {
    int choice, p;
    while (1) {
        printf("\n--- Dining Philosophers ---\n");
        printf("1. Eat\n2. Put Forks\n3. Back to Main Menu\nEnter choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter philosopher number (0–4): ");
                scanf("%d", &p);
                if (p >= 0 && p < 5) dp_take_forks(p);
                else printf("Invalid philosopher\n");
                break;
            case 2:
                printf("Enter philosopher number (0–4): ");
                scanf("%d", &p);
                if (p >= 0 && p < 5) dp_put_forks(p);
                else printf("Invalid philosopher\n");
                break;
            case 3:
                return;
            default:
                printf("Invalid choice\n");
        }
    }
}

// ---------- Producer-Consumer ----------
int buffer[5], in = 0, out = 0;
int mutex = 1, full = 0, empty = 5;

int wait(int s) { return --s; }
int signal(int s) { return ++s; }

void producer() {
    if (empty == 0) {
        printf("Buffer is Full. Producer waits...\n");
        return;
    }
    if (mutex == 1) {
        mutex = wait(mutex);
        full = signal(full);
        empty = wait(empty);
        buffer[in] = rand() % 100;
        printf("Produced: %d at %d\n", buffer[in], in);
        in = (in + 1) % 5;
        mutex = signal(mutex);
    }
}

void consumer() {
    if (full == 0) {
        printf("Buffer is Empty. Consumer waits...\n");
        return;
    }
    if (mutex == 1) {
        mutex = wait(mutex);
        full = wait(full);
        empty = signal(empty);
        printf("Consumed: %d from %d\n", buffer[out], out);
        out = (out + 1) % 5;
        mutex = signal(mutex);
    }
}
```

```
void run_producer_consumer() {
    int choice;
    while (1) {
        printf("\n--- Producer-Consumer ---\n");
        printf("1. Produce\n2. Consume\n3. Back to Main Menu\nEnter choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1: producer(); break;
            case 2: consumer(); break;
            case 3: return;
            default: printf("Invalid choice\n");
        }
    }
}

int main() {
    int main_choice;
    while (1) {
        printf("\n======= Main Menu =======\n");
        printf("1. Dining Philosophers\n2. Producer-Consumer\n3. Exit\nEnter your choice: ");
        scanf("%d", &main_choice);
        switch (main_choice) {
            case 1: run_dining_philosophers(); break;
            case 2: run_producer_consumer(); break;
            case 3: exit(0);
            default: printf("Invalid choice\n");
        }
    }
    return 0;
}
```

OUTPUT:



```
Shashank U ~ main ? v9.2.0 17:35 gcc os_4.c && .\a.exe
===== Main Menu =====
1. Dining Philosophers
2. Producer-Consumer
3. Exit
Enter your choice: 2

-- Producer-Consumer --
1. Produce
2. Consume
3. Back to Main Menu
Enter choice: 1
Produced: 41 at 0

-- Producer-Consumer --
1. Produce
2. Consume
3. Back to Main Menu
Enter choice: 2
Consumed: 41 from 0

-- Producer-Consumer --
1. Produce
2. Consume
3. Back to Main Menu
Enter choice: 2
Buffer is Empty. Consumer waits...

-- Producer-Consumer --
1. Produce
2. Consume
3. Back to Main Menu
Enter choice: 1
Produced: 67 at 1

-- Producer-Consumer --
1. Produce
2. Consume
3. Back to Main Menu
Enter choice: 3

===== Main Menu =====
1. Dining Philosophers
2. Producer-Consumer
3. Exit
Enter your choice: 2

-- Producer-Consumer --
1. Produce
2. Consume
3. Back to Main Menu
Enter choice: 3
```



```
===== Main Menu =====
1. Dining Philosophers
2. Producer-Consumer
3. Exit
Enter your choice: 1

-- Dining Philosophers --
1. Eat
2. Put Forks
3. Back to Main Menu
Enter choice: 1
Enter philosopher number (0 to 4): 0
Philosopher 0 is Eating

-- Dining Philosophers --
1. Eat
2. Put Forks
3. Back to Main Menu
Enter choice: 1
Enter philosopher number (0 to 4): 2
Philosopher 2 is Eating

-- Dining Philosophers --
1. Eat
2. Put Forks
3. Back to Main Menu
Enter choice: 2
Enter philosopher number (0 to 4): 1
Philosopher 1 was not Eating

-- Dining Philosophers --
1. Eat
2. Put Forks
3. Back to Main Menu
Enter choice: 3

===== Main Menu =====
1. Dining Philosophers
2. Producer-Consumer
3. Exit
Enter your choice: 3
```

Lab Program 5:

Write a C program to simulate: a) Bankers' algorithm for the purpose of deadlock avoidance.

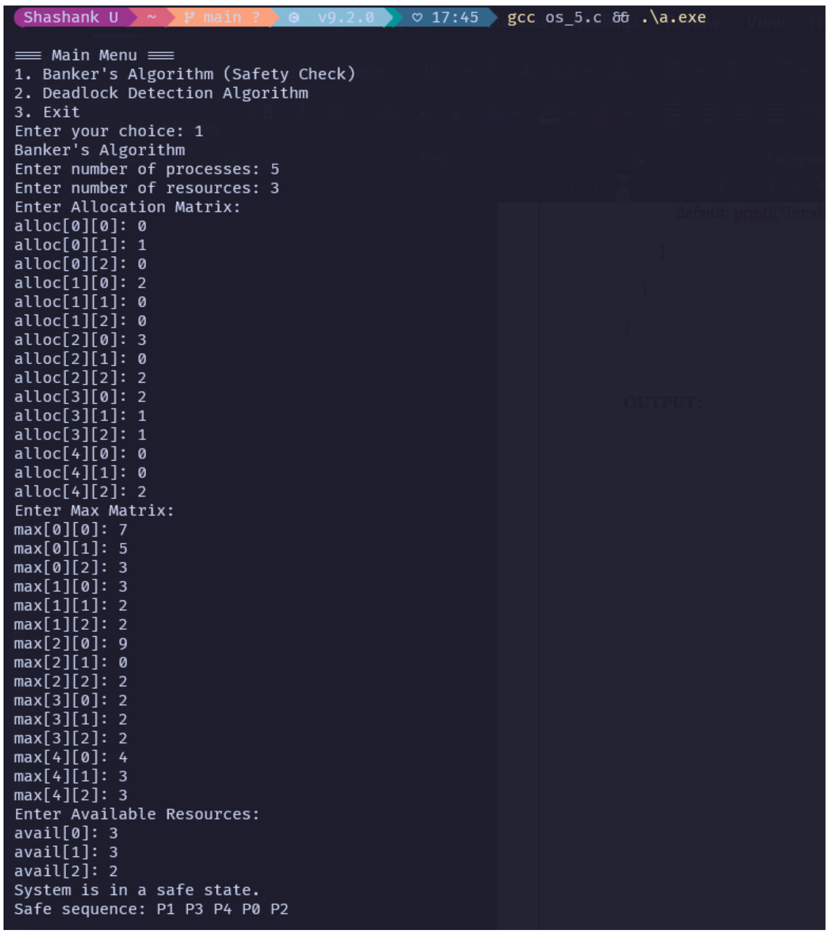
c) Deadlock Detection

```
#include <stdio.h>
#include <stdbool.h>

void bankers_algorithm() {
    int n, r;
    printf("Banker's Algorithm\n");
    printf("Enter number of processes: ");
    scanf("%d", &n);
    printf("Enter number of resources: ");
    scanf("%d", &r);

    int alloc[n][r], max[n][r], need[n][r], avail[r];

    printf("Enter Allocation Matrix:\n");
    for(int i=0; i<n; i++){
        for(int j=0; j<r; j++){
            printf("alloc[%d][%d]: ", i, j);
            scanf("%d", &alloc[i][j]);
        }
    }

    printf("Enter Max Matrix:\n");
    for(int i=0; i<n; i++){
        for(int j=0; j<r; j++){
            printf("max[%d][%d]: ", i, j);
            scanf("%d", &max[i][j]);
            need[i][j] = max[i][j] - alloc[i][j];
        }
```

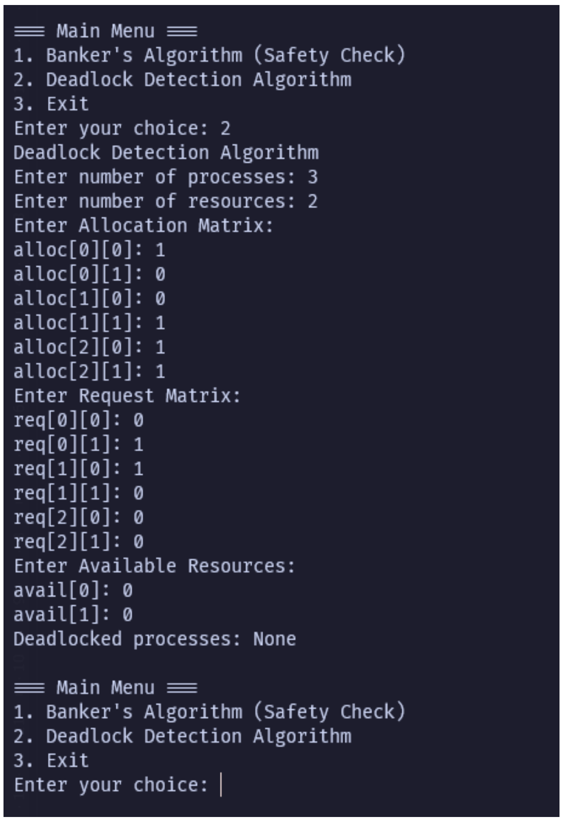
```
}

printf("Enter Available Resources:\n");
for(int i=0; i<r; i++){
    printf("avail[%d]: ", i);
    scanf("%d", &avail[i]);
}

bool finish[n];
for(int i=0; i<n; i++) finish[i] = false;

int safe_seq[n], count = 0;

while(count < n){
    bool found = false;
    for(int i=0; i<n; i++){
        if(!finish[i]){
            bool can_allocate = true;
            for(int j=0; j<r; j++){
                if(need[i][j] > avail[j]){
                    can_allocate = false;
                    break;
                }
            }
            if(can_allocate){
                for(int j=0; j<r; j++){
                    avail[j] += alloc[i][j];
                }
                safe_seq[count++] = i;
                finish[i] = true;
                found = true;
            }
            }
        }
        if(!found){
            printf("System is not in a safe state.\n");
            return;
        }
    }

    printf("System is in a safe state.\nSafe sequence: ");
    for(int i=0; i<n; i++){
        printf("P%d ", safe_seq[i]);
    }
    printf("\n");
}

void deadlock_detection() {
    int n, r;
    printf("Deadlock Detection Algorithm\n");
    printf("Enter number of processes: ");
    scanf("%d", &n);
    printf("Enter number of resources: ");
    scanf("%d", &r);

    int alloc[n][r], req[n][r], avail[r];

    printf("Enter Allocation Matrix:\n");
    for(int i=0; i<n; i++){
        for(int j=0; j<r; j++){
            printf("alloc[%d][%d]: ", i, j);
            scanf("%d", &alloc[i][j]);
        }
    }
```

```
printf("Enter Request Matrix:\n");
for(int i=0; i<n; i++){
    for(int j=0; j<r; j++){
        printf("req[%d][%d]: ", i, j);
        scanf("%d", &req[i][j]);
    }
}

printf("Enter Available Resources:\n");
for(int i=0; i<r; i++){
    printf("avail[%d]: ", i);
    scanf("%d", &avail[i]);
}

bool finish[n];
for(int i=0; i<n; i++) finish[i] = false;

// Repeat at most n times
for(int k=0; k<n; k++){
    bool allocated = false;
    for(int i=0; i<n; i++){
        if(!finish[i]){
            bool can_allocate = true;
            for(int j=0; j<r; j++){
                if(req[i][j] > avail[j]){
                    can_allocate = false;
                    break;
                }
            }
            if(can_allocate){
                for(int j=0; j<r; j++){
                    avail[j] += alloc[i][j];
                }
                finish[i] = true;
                allocated = true;
            }
        }
    }
    if(!allocated) break;
  }

  printf("Deadlocked processes: ");
  bool deadlock_found = false;
  for(int i=0; i<n; i++){
      if(!finish[i]){
          printf("P%d ", i);
          deadlock_found = true;
      }
  }
  if(!deadlock_found) printf("None");
  printf("\n");
}

int main() {
  int choice;
  while(1){
      printf("\n=== Main Menu ===\n");
      printf("1. Banker's Algorithm (Safety Check)\n");
      printf("2. Deadlock Detection Algorithm\n");
      printf("3. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);
      switch(choice){
            case 1: bankers_algorithm(); break;

            case 2: deadlock_detection(); break;

            case 3: return 0;

            default: printf("Invalid choice, try again.\n");

        }

    }

}
```

OUTPUT:



```
Shashank U  ~  main ?  v9.2.0  17:45  gcc os_5.c && .\a.exe

══ Main Menu ══
1. Banker's Algorithm (Safety Check)
2. Deadlock Detection Algorithm
3. Exit
Enter your choice: 1
Banker's Algorithm
Enter number of processes: 5
Enter number of resources: 3
Enter Allocation Matrix:
alloc[0][0]: 0
alloc[0][1]: 1
alloc[0][2]: 0
alloc[1][0]: 2
alloc[1][1]: 0
alloc[1][2]: 0
alloc[2][0]: 3
alloc[2][1]: 0
alloc[2][2]: 2
alloc[3][0]: 2
alloc[3][1]: 1
alloc[3][2]: 1
alloc[4][0]: 0
alloc[4][1]: 0
alloc[4][2]: 2
Enter Max Matrix:
max[0][0]: 7
max[0][1]: 5
max[0][2]: 3
max[1][0]: 3
max[1][1]: 2
max[1][2]: 2
max[2][0]: 9
max[2][1]: 0
max[2][2]: 2
max[3][0]: 2
max[3][1]: 2
max[3][2]: 2
max[4][0]: 4
max[4][1]: 3
max[4][2]: 3
Enter Available Resources:
avail[0]: 3
avail[1]: 3
avail[2]: 2
System is in a safe state.
Safe sequence: P1 P3 P4 P0 P2
```



```
══ Main Menu ══
1. Banker's Algorithm (Safety Check)
2. Deadlock Detection Algorithm
3. Exit
Enter your choice: 2
Deadlock Detection Algorithm
Enter number of processes: 3
Enter number of resources: 2
Enter Allocation Matrix:
alloc[0][0]: 1
alloc[0][1]: 0
alloc[1][0]: 0
alloc[1][1]: 1
alloc[2][0]: 1
alloc[2][1]: 1
Enter Request Matrix:
req[0][0]: 0
req[0][1]: 1
req[1][0]: 1
req[1][1]: 0
req[2][0]: 0
req[2][1]: 0
Enter Available Resources:
avail[0]: 0
avail[1]: 0
Deadlocked processes: None

══ Main Menu ══
1. Banker's Algorithm (Safety Check)
2. Deadlock Detection Algorithm
3. Exit
Enter your choice: |
```

Lab Program 6

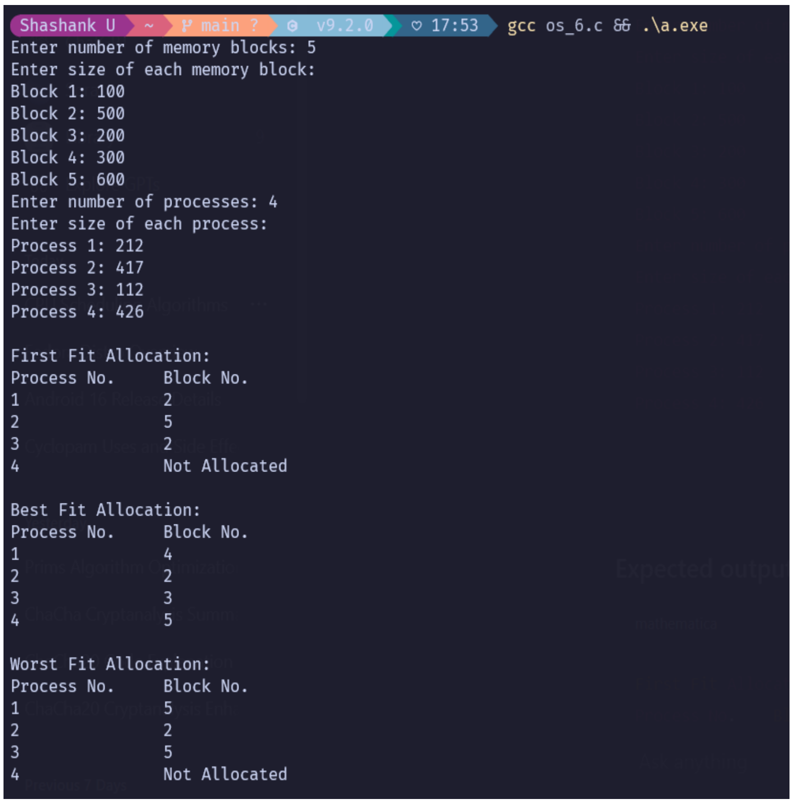
Write a C program to simulate the following contiguous memory allocation techniques.

a) Worst-fit b) Best-fit c) First-fit

```
#include <stdio.h>

#include <stdlib.h>


#define MAX_BLOCKS 20

#define MAX_PROCESSES 20


void print_allocation(int allocation[], int np) {

    printf("Process No.\tBlock No.\n");

    for (int i = 0; i < np; i++) {

        printf("%d\t\t", i + 1);

        if (allocation[i] != -1)

            printf("%d\n", allocation[i] + 1); // block number (1-based)

        else

            printf("Not Allocated\n");

    }

}


// First Fit Memory Allocation

void first_fit(int blockSize[], int nb, int processSize[], int np) {

    int allocation[MAX_PROCESSES];

    for (int i = 0; i < np; i++)

        allocation[i] = -1;


    int tempBlockSize[MAX_BLOCKS];

    for (int i = 0; i < nb; i++)

        tempBlockSize[i] = blockSize[i];


    for (int i = 0; i < np; i++) {
        for (int j = 0; j < nb; j++) {
            if (tempBlockSize[j] >= processSize[i]) {
                allocation[i] = j;
                tempBlockSize[j] -= processSize[i];
                break;
            }
        }
    }
    printf("\nFirst Fit Allocation:\n");
    print_allocation(allocation, np);
}

// Best Fit Memory Allocation
void best_fit(int blockSize[], int nb, int processSize[], int np) {
    int allocation[MAX_PROCESSES];
    for (int i = 0; i < np; i++)
        allocation[i] = -1;

    int tempBlockSize[MAX_BLOCKS];
    for (int i = 0; i < nb; i++)
        tempBlockSize[i] = blockSize[i];

    for (int i = 0; i < np; i++) {
        int bestIdx = -1;
        for (int j = 0; j < nb; j++) {
            if (tempBlockSize[j] >= processSize[i]) {
                if (bestIdx == -1 || tempBlockSize[j] < tempBlockSize[bestIdx])
                    bestIdx = j;
            }
        }
        if (bestIdx != -1) {
            allocation[i] = bestIdx;
            tempBlockSize[bestIdx] -= processSize[i];
        }
    }
    printf("\nBest Fit Allocation:\n");
    print_allocation(allocation, np);
}

// Worst Fit Memory Allocation
void worst_fit(int blockSize[], int nb, int processSize[], int np) {
    int allocation[MAX_PROCESSES];
    for (int i = 0; i < np; i++)
        allocation[i] = -1;

    int tempBlockSize[MAX_BLOCKS];
    for (int i = 0; i < nb; i++)
        tempBlockSize[i] = blockSize[i];

    for (int i = 0; i < np; i++) {
        int worstIdx = -1;
        for (int j = 0; j < nb; j++) {
            if (tempBlockSize[j] >= processSize[i]) {
                if (worstIdx == -1 || tempBlockSize[j] > tempBlockSize[worstIdx])
                    worstIdx = j;
            }
        }
        if (worstIdx != -1) {
            allocation[i] = worstIdx;
            tempBlockSize[worstIdx] -= processSize[i];
        }
    }
    printf("\nWorst Fit Allocation:\n");
    print_allocation(allocation, np);
```

```
}


int main() {
    int blockSize[MAX_BLOCKS], processSize[MAX_PROCESSES];
    int nb, np;


    printf("Enter number of memory blocks: ");
    scanf("%d", &nb);
    printf("Enter size of each memory block:\n");
    for (int i = 0; i < nb; i++) {
        printf("Block %d: ", i + 1);
        scanf("%d", &blockSize[i]);
    }
    printf("Enter number of processes: ");
    scanf("%d", &np);
    printf("Enter size of each process:\n");
    for (int i = 0; i < np; i++) {
        printf("Process %d: ", i + 1);
        scanf("%d", &processSize[i]);
    }


    first_fit(blockSize, nb, processSize, np);
    best_fit(blockSize, nb, processSize, np);
    worst_fit(blockSize, nb, processSize, np);


    return 0;
}
```

OUTPUT:



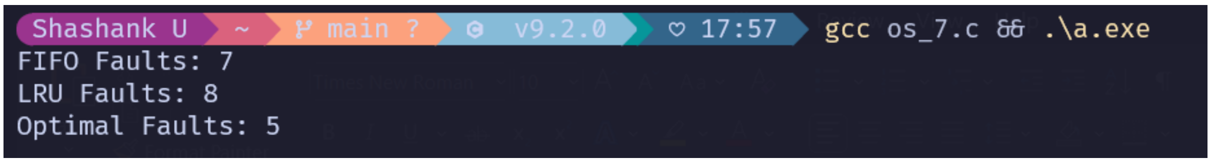
```
Shashank U  ~  main ?   v9.2.0   17:53  gcc os_6.c && .\a.exe
Enter number of memory blocks: 5
Enter size of each memory block:
Block 1: 100
Block 2: 500
Block 3: 200
Block 4: 300
Block 5: 600
Enter number of processes: 4
Enter size of each process:
Process 1: 212
Process 2: 417
Process 3: 112
Process 4: 426

First Fit Allocation:
Process No.     Block No.
1               2
2               5
3               2
4               Not Allocated

Best Fit Allocation:
Process No.     Block No.
1               4
2               2
3               3
4               5

Worst Fit Allocation:
Process No.     Block No.
1               5
2               2
3               5
4               Not Allocated
```

Lab Program 7

Write a C program to simulate page replacement algorithms. a) FIFO b) LRU c) Optimal

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 20

void FIFO(int pages[], int n, int frames) {
    int queue[MAX], front = 0, rear = 0, count = 0, faults = 0;
    for (int i = 0; i < n; i++) {
        int found = 0;
        for (int j = front; j < rear; j++)
            if (queue[j % frames] == pages[i]) found = 1;
        if (!found) {
            if (count < frames) count++;
            queue[rear % frames] = pages[i];
            rear++;
            if (rear - front > frames) front++;
            faults++;
        }
    }
    printf("FIFO Faults: %d\n", faults);
}

void LRU(int pages[], int n, int frames) {
    int frame[MAX], time[MAX], faults = 0, t = 0;
    for (int i = 0; i < frames; i++) frame[i] = -1;
    for (int i = 0; i < n; i++) {
        int found = 0;
        for (int j = 0; j < frames; j++) {
            if (frame[j] == pages[i]) {
                found = 1;
                time[j] = ++t;
            }
        }
        if (!found) {
            int min = 0;
            for (int j = 1; j < frames; j++)
                if (time[j] < time[min]) min = j;
            frame[min] = pages[i];
            time[min] = ++t;
            faults++;
        }
    }
    printf("LRU Faults: %d\n", faults);
}

void Optimal(int pages[], int n, int frames) {
    int frame[MAX], faults = 0;
    for (int i = 0; i < frames; i++) frame[i] = -1;
    for (int i = 0; i < n; i++) {
        int found = 0;
        for (int j = 0; j < frames; j++)
            if (frame[j] == pages[i]) found = 1;
        if (!found) {
            int idx = -1, farthest = i + 1;
            for (int j = 0; j < frames; j++) {
                int k;
                for (k = i + 1; k < n; k++)
                    if (frame[j] == pages[k]) break;
                if (k > farthest) {
                    farthest = k;
                    idx = j;
                }
                if (k == n) {
                    idx = j;
                    break;
                }
            }
            if (idx == -1) idx = 0;
            frame[idx] = pages[i];
            faults++;
        }
    }
    printf("Optimal Faults: %d\n", faults);
}

void main() {
    int pages[] = {1, 2, 3, 2, 4, 1, 5, 2};
    int n = sizeof(pages) / sizeof(pages[0]);
    int frames = 3;

    FIFO(pages, n, frames);
    LRU(pages, n, frames);
    Optimal(pages, n, frames);
}
```

OUTPUT:



```
Shashank U  ~  main ?  v9.2.0  17:57  gcc os_7.c && .\a.exe
FIFO Faults: 7
LRU Faults: 8
Optimal Faults: 5
```

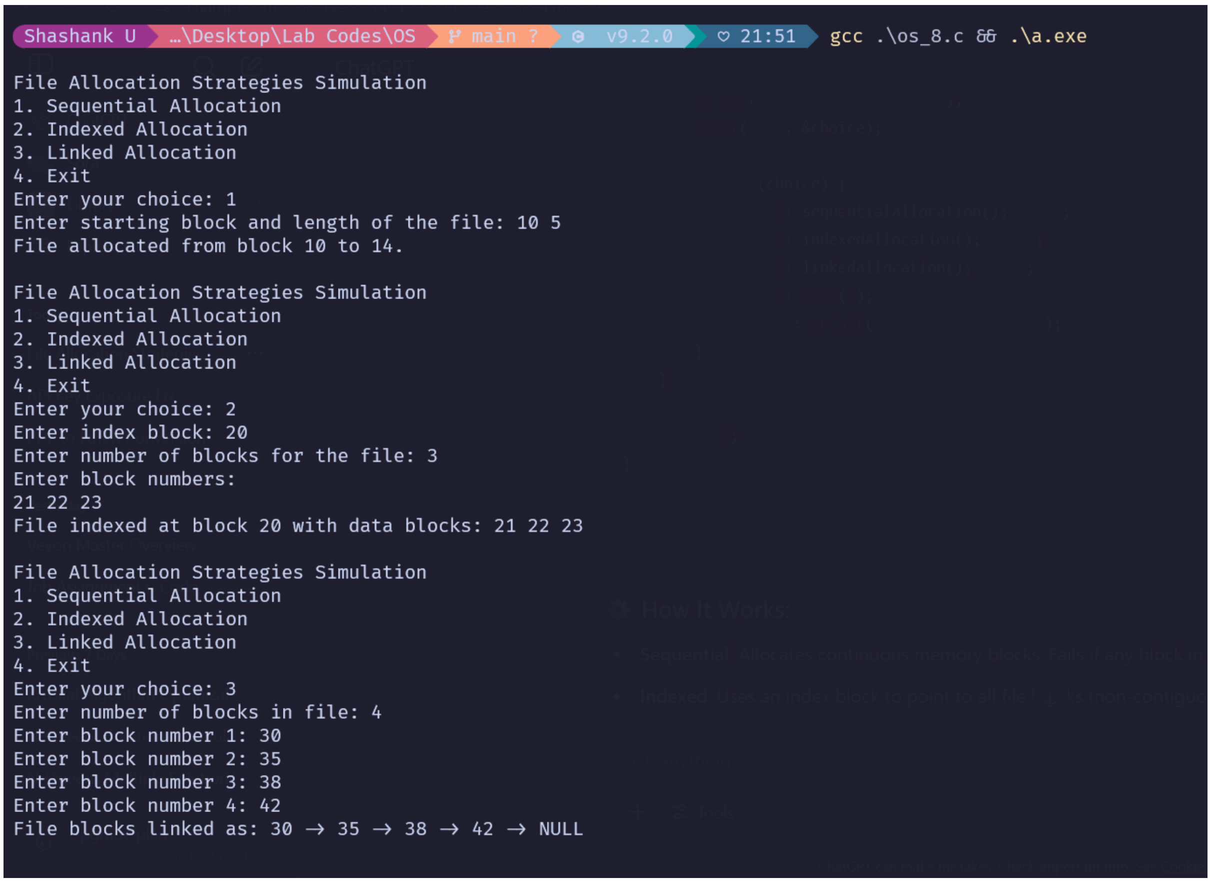
Lab Program 8

Write a C program to simulate the following file allocation strategies. a) Sequential b) Indexed c) Linked

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


#define MAX_BLOCKS 100

#define MAX_FILES 10


int memory[MAX_BLOCKS]; // 0 means free, 1 means allocated
```

```
// Struct for Linked Allocation
typedef struct Node {
    int block;
    struct Node* next;
} Node;

// Utility to reset memory
void resetMemory() {
    for (int i = 0; i < MAX_BLOCKS; i++)
        memory[i] = 0;
}

// a) Sequential Allocation
void sequentialAllocation() {
    resetMemory();
    int start, length;

    printf("Enter starting block and length of the file: ");
    scanf("%d%d", &start, &length);

    // Check if within memory and free
    if (start < 0 || start + length > MAX_BLOCKS) {
        printf("Invalid input. Out of bounds.\n");
        return;
    }

    for (int i = start; i < start + length; i++) {
        if (memory[i]) {
            printf("Block %d already allocated. Cannot allocate.\n", i);
            return;
        }
    }
    for (int i = start; i < start + length; i++) {
        memory[i] = 1;
    }

    printf("File allocated from block %d to %d.\n", start, start + length - 1);
}

// b) Indexed Allocation
void indexedAllocation() {
    resetMemory();
    int indexBlock, numBlocks, blocks[MAX_BLOCKS];

    printf("Enter index block: ");
    scanf("%d", &indexBlock);

    if (indexBlock < 0 || indexBlock >= MAX_BLOCKS || memory[indexBlock]) {
        printf("Index block invalid or already allocated.\n");
        return;
    }

    printf("Enter number of blocks for the file: ");
    scanf("%d", &numBlocks);

    printf("Enter block numbers:\n");
    for (int i = 0; i < numBlocks; i++) {
        scanf("%d", &blocks[i]);
        if (blocks[i] < 0 || blocks[i] >= MAX_BLOCKS || memory[blocks[i]]) {
            printf("Block %d invalid or already allocated.\n", blocks[i]);
            return;
        }
    }
    // Allocate
    memory[indexBlock] = 1;
    for (int i = 0; i < numBlocks; i++)
        memory[blocks[i]] = 1;

    printf("File indexed at block %d with data blocks: ", indexBlock);
    for (int i = 0; i < numBlocks; i++)
        printf("%d ", blocks[i]);
    printf("\n");
}

// c) Linked Allocation
void linkedAllocation() {
    resetMemory();
    int numBlocks, block;
    Node *head = NULL, *temp = NULL, *newNode = NULL;

    printf("Enter number of blocks in file: ");
    scanf("%d", &numBlocks);

    for (int i = 0; i < numBlocks; i++) {
        printf("Enter block number %d: ", i + 1);
        scanf("%d", &block);

        if (block < 0 || block >= MAX_BLOCKS || memory[block]) {
            printf("Block %d invalid or already allocated.\n", block);
            return;
        }

        memory[block] = 1;
        newNode = (Node*)malloc(sizeof(Node));
        newNode->block = block;
        newNode->next = NULL;

        if (head == NULL)
            head = newNode;
        else
            temp->next = newNode;

        temp = newNode;
    }

    printf("File blocks linked as: ");
    temp = head;
    while (temp) {
        printf("%d -> ", temp->block);
        Node* toFree = temp;
        temp = temp->next;
        free(toFree); // cleanup
    }
    printf("NULL\n");
}

// Main Menu
int main() {
    int choice;

    while (1) {
        printf("\nFile Allocation Strategies Simulation\n");
        printf("1. Sequential Allocation\n");
        printf("2. Indexed Allocation\n");
        printf("3. Linked Allocation\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: sequentialAllocation(); break;
            case 2: indexedAllocation(); break;
            case 3: linkedAllocation(); break;
            case 4: exit(0);
            default: printf("Invalid choice.\n");
        }
    }

    return 0;
}
```

OUTPUT:



```
Shashank U  …\Desktop\Lab Codes\OS  main ?  v9.2.0  21:51  gcc .\os_8.c && .\a.exe

File Allocation Strategies Simulation
1. Sequential Allocation
2. Indexed Allocation
3. Linked Allocation
4. Exit
Enter your choice: 1
Enter starting block and length of the file: 10 5
File allocated from block 10 to 14.

File Allocation Strategies Simulation
1. Sequential Allocation
2. Indexed Allocation
3. Linked Allocation
4. Exit
Enter your choice: 2
Enter index block: 20
Enter number of blocks for the file: 3
Enter block numbers:
21 22 23
File indexed at block 20 with data blocks: 21 22 23

File Allocation Strategies Simulation
1. Sequential Allocation
2. Indexed Allocation
3. Linked Allocation
4. Exit
Enter your choice: 3
Enter number of blocks in file: 4
Enter block number 1: 30
Enter block number 2: 35
Enter block number 3: 38
Enter block number 4: 42
File blocks linked as: 30 → 35 → 38 → 42 → NULL
```